The Rightmost Equal-Cost Position Problem

Maxime Crochemore[1,3], Alessio Langiu[1] and Filippo Mignosi[2]

[1] King's College London, London, UK
{*Maxime.Crochemore,Alessio.Langiu*}*@kcl.ac.uk*
[2] University of L'Aquila, L'Aquila, Italy
Filippo.Mignosi@di.univaq.it
[3] Université Paris-Est, France

Abstract

LZ77-based compression schemes compress the input text by replacing factors in the text with an encoded reference to a previous occurrence formed by the couple (length, offset). For a given factor, the smallest is the offset, the smallest is the resulting compression ratio. This is optimally achieved by using the rightmost occurrence of a factor in the previous text. Given a cost function, for instance the minimum number of bits used to represent an integer, we define the Rightmost Equal-Cost Position (REP) problem as the problem of finding one of the occurrences of a factor which cost is equal to the cost of the rightmost one. We present the Multi-Layer Suffix Tree data structure that, for a text of length n, at any time i, it provides REP(LPF) in constant time, where LPF is the longest previous factor, i.e. the greedy phrase, a reference to the list of REP({set of prefixes of LPF}) in constant time and REP(p) in time $O(|p| \log\log n)$ for any given pattern p.

Introduction

The Rightmost Equal-Cost Position (REP) is an occurrence, pointed out by its offset, w.r.t. the current position, which reference uses a number of bits that is equal to the number of bits of the offset of the rightmost occurrence

of a factor (substring) in the previous text. This problem mainly comes from the data compression field and particularly from the LZ77 based schemes, where the length and offset pair are used to encode a dictionary phrase.

The foundational Ziv and Lempel LZ77 algorithm [22] is the basis of almost all the famous dictionary compressors, like gZip, PkZip, WinZip and 7Zip. They consider a portion of the previous text as the dictionary, i.e. they use a dynamic dictionary formed by the set of all the factors of the text up to the current position within a sliding window of fixed size. A dictionary phrase refers to an occurrence of such phrase in the text by using the couple (length, offset), where the offset is the backward offset w.r.t. the current position. Since a phrase is usually repeated more than once along the text and since pointers with smaller offset have usually a smaller representation, the occurrence close to the current position is preferred.

Usually the length and offset pair is encoded by using a variable length code such as Huffman code or Elias Delta code (see for instance Deflate, gZip, PkZip, LZRW4, LZMA and 7zip descriptions in [16]). Assuming that the lengths of encoded values is a monotonic non decreasing function, it is straightforward that the smaller is the value, the smaller is its encoded cost, i.e. the length in bits of the encoded value. We define a Rightmost Equal-Cost Position (REP) of w as the one of the occurrences of the factor w which cost is the same as the rightmost one, according to a fixed cost function. The REP values can be optimally used to represent a dictionary phrase instead of the rightmost occurrence. The cost function can be, for instance, the number of bits of the binary representation, i.e. the smaller integer greater than the $\log_2(x)$, where x is the value we have to encode. Later on in this paper we show how an example using the bit length of the Elias Gamma code as cost function. Furthermore, in LZ77 based compression, the greedy approach is used to parse the text into phrases, i.e, in an iterative way, the longest match between the dictionary and the forwarding text is chosen. This is commonly called the greedy phrase. Some LZ77-based algorithms as Deflate algorithm and the compressors based on them, like gZip and PkZip, use variants of the greedy approach to parse the text. Deflate64 algorithm implemented in WinZip and 7zip, contains some heuristics to parse differently the text in order to improve the compression ratio, but its time complexity was never clearly stated.

The research about parsing optimality for dictionary compression produced in the last decades some noticeable results (see for instance [2, 8, 10, 12, 13, 20]). The greedy parsing is optimal for LZ77 dynamic dictionary

schemes in the case of fixed length encoding of length and offset pair values as showed in [2, 20], but it is not optimal, in terms of compression performance, when variable length code (VLC) are used to encode the pointers to dictionary phrases. Recently, some optimal parsing solutions have been presented for dynamic dictionary and VLC, see [3, 4, 5]. In [4] presented a solution for LZ77 dictionary and VLC, called Dictionary-Symbolwise Flexible Parsing. It uses a graph-based model for the parsing problem where each node represent a position in the text and edges represent dictionary phrases. Edges are weighted according to the bit length of the encoded length and offset pair. For details about this algorithm we refer to the original paper [4]. Unfortunately, this latter paper lacks of a practical solution for the problem of finding the smallest offset (the rightmost position) for a dictionary phrase. Even more, optimal parsing algorithms for the VLC case need to know, for any position in the text, the length and offset pairs of all the prefixes of the Longest Previous Factor (LPF) or, at least, those prefixes which occurrence have a different offset.

The main goal of this work is to state the problem about the Rightmost Equal-cost Position (REP) and to present the Multilayer Suffix Tree as an efficient solution. In Section 1 we show how to reduce the problem of finding the rightmost position of a word w to the weaker REP(w) problem. In Section 2 we present the new Multilayer Suffix Tree full-text index data structure that uses $O(n \log n)$ amortized building time and linear space. In Section 3 we show how to solve the REP problem for any pattern p in $O(|p| \log \log n)$ time. Finally, in Section 4 we show how to use this new data structure, at any time i, to retrieve REP(LPF) and a pointer to the list of REP(SPF) in constant time, where LPF is the longest previous factor starting at position i and SPF is the set of prefixes of LPF, and, furthermore, we show some experimental results.

1 Definition of the Problem

Let $Pos(w) \subset \mathbb{N}$ the set of all the occurrences of $w \in Fact(T)$ in the text $T \in \Sigma^*$, where $Fact(T)$ is the set of the factors of T. Let $Offset(w) \subset \mathbb{N}$ be the set of all the occurrence offsets of $w \in Fact(T)$ in the text T, i.e. $x \in Offset(w)$ *iff* x is the distance between the position of an occurrence of w and the end of the text T. For instance, given the text $T = babcabbabab$ of length $|T| = 12$ and the factor $w = abb$ of length $|w| = 3$, the set of

positions of w over T is $Pos(w) = \{4, 9\}$. The set of the offsets of w over T is $\mathit{Offset}(w) = \{7, 2\}$. Notice that $x \in \mathit{Offset}(w)$ *iff* exists $y \in Pos(w)$ such that $x = |T| - y - 1$. Since the offsets are function of positions, there is a bijection between $Pos(w)$ and $\mathit{Offset}(w)$, for any factor w.

Given a number encoding method, let $\mathit{Bitlen} : \mathbb{N} \to \mathbb{N}$ a function that associates to a number x the length in bit of the encoding of x. Let us consider the equivalence relation *having equal codeword bit-length* on the set $\mathit{Offset}(w)$. The numbers $x, y \in \mathit{Offset}(w)$ are bit-length equivalent *iff* $\mathit{Bitlen}(x) = \mathit{Bitlen}(y)$. Let us notice that the *having equal codeword bit-length* relation induces a partition on $\mathit{Offset}(w)$.

Definition 1.1. Given a word $w \in \Sigma^*$, a text $T = a_1 \dots a_n \in \Sigma^*$, the *rightmost* occurrence of w over T is the occurrence of w that appears closest to the end of the text, if w appears at least once in the text T, otherwise it is not defined. More formally,

$$\mathit{rightmost}(w) = \begin{cases} \min\{x \mid x \in \mathit{Offset}(w)\} & \text{if } \mathit{Offset}(w) \neq \emptyset \\ \text{not defined} & \text{if } \mathit{Offset}(w) = \emptyset \end{cases}$$

Let us notice that referring to the rightmost occurrence of a word in a dynamic setting, where the input text is processed left to right, corresponds to referring to the rightmost occurrence over the text *already processed*. Indeed, if at a certain algorithm step we have processed the first i symbols of the text, the rightmost occurrence of w is the occurrence of w closest to the position i of the text.

Definition 1.2. Let $\mathit{rightmost}_i(w)$ be the rightmost occurrence of w over T_i, where T_i is the prefix of the text T ending at the position i in T. Obviously, $\mathit{rightmost}_n(w) = \mathit{rightmost}(w)$ for $|T| = n$.

In many practical data compression algorithms the text we are able to refer to is just a portion of the whole text. Let $T[j : i]$ be the factor of the text T starting from the position j and ending to the position i. We generalize the definition of $\mathit{rightmost}(w)$ over a factor $T[j : i]$ of T as follows.

Definition 1.3. Let $\mathit{rightmost}_{j,i}(w)$ be the rightmost occurrence of w over $T[j : i]$, where $T[j : i]$ is the factor of the text T starting at the position j and ending at the position i of length $i - j + 1$. Obviously, $\mathit{rightmost}_{1,n}(w) = \mathit{rightmost}(w)$ for $|T| = n$.

Definition 1.4. The Rightmost Equal-Cost Position (REP). Let us suppose that we have a text $T \in \Sigma^*$, a pattern $w \in \Sigma^*$ and a point i in time. If w appears at least once in T_i, then a *rightmost equal-cost position* of w, REP(w), over T_i is a position j in the text which offset is in $[rightmost_i(w)]$, where $[rightmost_i(w)]$ is the equivalence class induced by the relation *having equal codeword bit-length* containing the element $rightmost_i(w)$. Otherwise, when w does not appear in T_i, the REP(w) = 0.

2 The Multilayer Suffix Tree

The main idea of this new data structure, is based on two observations. The first one is that the equivalence relation *having equal codeword bit-length* induces a partition on *Offset*(w), for any w, and also induces a partition on the set of all the possible offsets over a text T independently from any specific factor, i.e. on the set $[1..|T|]$. The second observation is that for any encoding method for which the *Bitlen* function is a monotonic function, each equivalence class in $[1..|T|]$ is composed by contiguous points in $[1..|T|]$. Indeed, given a point $p \in [1..|T|]$, the equivalence class $[p]$ is equal to the set $[j..i]$, with $j \leq p \leq i$, $j = \min\{x \in [p]\}$ and $i = \max\{x \in [p]\}$.

Assuming the *Bitlen* function as the cost function of the dictionary phrases, we exploit the discreteness of *Bitlen* in order to solve the REP problem by using a set of suffix trees for sliding window, each one devoted to one or more classes of equivalence of the relation *having equal code bit-length*.

Fixed an encoding method for numbers and a text T, we assume that *Bitlen* is a monotonic function and that we know the set $B = \{b_1, b_2, \ldots, b_s\}$, with $b_1 < b_2 < \cdots < b_s$, that is the set of the *Bitlen* values of all the possible offsets over T. We define the set $SW = \{sw_1, sw_2, \ldots, sw_s\}$, with $sw_1 < sw_2 < \cdots < sw_s$, where sw_i is the greatest integer (smaller than or equal to the length of the text T) such that $Bitlen(j)$ is less that or equal to b_i. More formally, $sw_i = \max\{j \leq |T| \mid Bitlen(j) \leq b_i\}$. Notice that $sw_s = |T|$.

All the standard non-unary representation of numbers satisfy the following property.

Property 2.1. *There exists a constant $k > 1$ and an integer $\hat{k}$ such that for any $\hat{k} \leq i < s$ one has $sw_i \geq k\, sw_{i-1}$.*

The Huffman code do not strictly satisfy above property, but it easy to find a function $f(w)$ always greater than *Bitlen*(Huffman code of w) that is a good approximation of it and it does have the Property 2.1 or, alternatively, you can impose that property rearranging the Huffman trees.

Let us consider an example where integers are encoded by using the Elias γ codes. The Table 1 reports the Elias γ codes and the *Bitlen* values for integers from 1 to 18. Suppose, for instance, that our cost function is associated to this Elias codes and that we have a text T of length 18. The set B therefore is $B = \{b_1 = 1, b_2 = 3, b_3 = 5, b_4 = 7, b_5 = 9\}$ and we have that $sw_1 = 1$, $sw_2 = 3$, $sw_3 = 7$, $sw_4 = 15$ and $sw_5 = 18$. Notice that, indeed, Property 2.1 is satisfied for $\hat{k} = 2$ and $k = 2$.

Let us now introduce the Multilayer Suffix Tree data structure. We suppose that a text T is provided *online* and at time i the first i characters of T have been read, i.e. at time i the prefix T_i of length i of the text T has been

i	$\gamma(i)$	$Bitlen(\gamma(i))$
1	1	1
2	010	3
3	011	3
4	00100	5
5	00101	5
6	00110	5
7	00111	5
8	0001000	7
9	0001001	7
10	0001010	7
11	0001011	7
12	0001100	7
13	0001101	7
14	0001110	7
15	0001111	7
16	000010000	9
17	000010001	9
18	000010010	9

Table 1: Elias γ code for integers from 1 to 18 and their *Bitlen* value.

read.

Definition 2.1. The Multilayer Suffix Tree is a data structure composed by the set $S = \{S_{sw_1}, S_{sw_2}, \ldots, S_{sw_s}\}$ of suffix trees where, for any $\alpha \in SW$ and at any moment i, S_α is the suffix tree for sliding window of T_i with sliding window of size α such that S_α represents all the factors of $T[i-\alpha : i]$. We call S_α simply the layer α or the layer of size α.

From now on we will refer to suffix trees or layers indifferently.

We use the *online* suffix tree for sliding window construction algorithm introduced by Larson in [11] and later refined by Senft in [17, 18], in order to build each layer of our multilayer suffix tree. Let us recall that in [11] an *online* and linear time construction for the suffix tree is reported. The suffix tree uses linear space w.r.t. the size of the sliding window. Therefore, for any $S_\alpha \in S = \{S_{sw_1}, S_{sw_2}, \ldots, S_{sw_s}\}$, S_α is a full-text index for $T[i-\alpha : i]$, where T is a given text and i is a point in time. S_α uses $O(\alpha)$ space.

In order to decrease the practical space requirement, we think that it is possible to adapt our data structure to work with other classic indexes for sliding window (see for instance [9, 14, 19]) or to use the recently presented compressed versions of the suffix tree (e.g. [7, 15]).

Proposition 2.2. 1. *If a pattern w is in layer α with $\alpha \in SW$, then w is also in layer β for any $\beta \in SW$ with $\alpha \leq \beta$.* 2. *If a pattern w is not in a layer α, $\alpha \in SW$, then w is not in layer β with $\beta \leq \alpha$*

Proof. The proof of the property at point 1 comes immediately from suffix trees properties. Since layer α is a full text index for $T[i-\alpha : i]$ and layer *beta* is a full-text index for $T[i-\beta : i]$, for any α, $\beta \in SW$ with $\alpha \leq \beta$ and for any i, $T[i-\alpha : i]$ is a suffix of $T[i-\beta : i]$. The property at point 2 can be deduced by point 1. □

Proposition 2.3. *Fixed a text T of size $|T| = n$, at any moment i with $0 \leq i \leq n$ and for any standard variable-length code, the multilayer suffix tree uses $O(i)$ space.*

Proof. Since at time i the maximum offset of all the occurrences in the text T_i is $O(i)$, for any standard variable-length code the maximum value of the set SW is $O(i)$. Since Property 2.1 holds for the set SW and since the multilayer suffix tree space is equal to the sum of the space of its layers, as

an immediate consequence we have that space used by the multilayer suffix tree is $O(\sum_{\alpha \in SW} \alpha) = O(i)$. □

For instance, at time i, if we consider the usual binary representation of numbers, the values $\alpha \in SW$ turn out to be powers of 2 from 1 to j, where j is the greatest power of 2 smaller than i, plus i, i.e. $SW = \{1, 2, 4, \ldots j, i\}$ and $\sum_{\alpha \in SW} \alpha < 2j + i < 3i$ and $O(\sum_{\alpha \in SW} \alpha) = O(i)$.

From the linear time of the *online* construction of the suffix tree for sliding window and since the number of layers is $|SW| = O(\log |T|)$, we can immediately state the following proposition.

Proposition 2.4. *Given a text T of length $|T| = n$, for any standard variable-length code, it is possible to build online the multilayer suffix tree in $O(n \log n)$ amortized time.*

3 Solving REP for a Generic Pattern

We want now to show how to answer to the REP(p) for a given pattern p of length $|p|$ and a text T_i of length i in $O(|p| \log \log i)$.

Proposition 3.1. *If a pattern p is in layer β and is not in layer α, where α is the maximum of the values in SW smaller than β, then any occurrence of p in the layer β correctly solve the REP problem for the pattern p.*

Proof. If a pattern p is in layer β and is not in layer α, where α is the maximum of the values in SW smaller than β, then from Prop. 2.2, it follows that β is the smallest layer where it appears p. Therefore p has at least one occurrence in $T[i - \beta : i - \alpha - 1]$, i.e. $rightmost_i(p) \in (\alpha..\beta]$. Since $(\alpha..\beta]$ is the equivalence class $[\beta]$ of the *having equal codeword bit-length* relation, we have that any occurrence of p in the layer β correctly solve the REP problem for p. □

Remark. Let us notice that if S_{sw_x} is the smallest layer where $rightmost_i(w)$ appears, then the *Bitlen* value of the offset of $rightmost_i(w)$ is equal to b_x.

Using above proposition, we are able to solve the problem REP(p) once we find the smallest layer containing the rightmost occurrence of the pattern, if any, otherwise we just report 0.

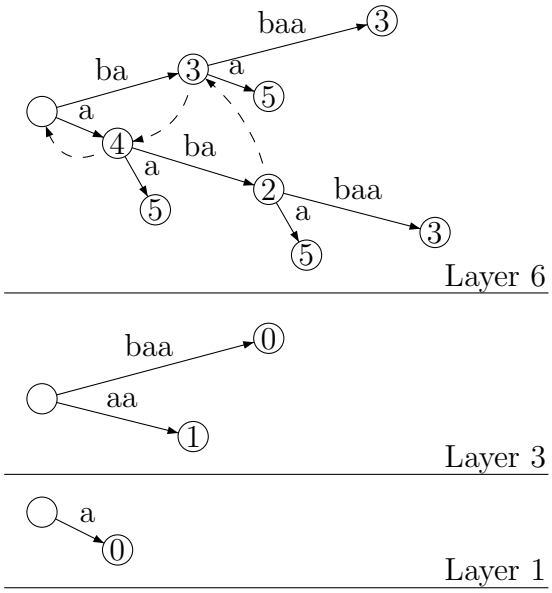

Figure 1: The Multilayer Suffix Tree for the text $T =$ `ababaa` and for the Elias γ-code, where $B = \{1, 3, 5\}$, $SW = \{1, 3, 6\}$. The solid edges are regular ones and the dashed links are the suffix-links of internal nodes. For convenience, we added edge labels with the substring of T associated to edges. The node value is the position over the text of the incoming edge. We omitted the string depth of nodes. Let consider, for instance, the phrase $w =$ '*ba*" with $Pos(w) = \{1, 3\}$, $Offset(w) = \{4, 2\}$, $rightmost(w) = 2$ and $\gamma(2) =$*010*. Since w is in layer $sw_2 = 3$ and is not in layer $sw_1 = 1$, we have that $Bitlen(rightmost(w)) = 3$ is equal to $Bitlen(sw_2 = 3) = b_2 = 3$.

What follows is the trivial search of the smallest layer that contains an occurrence of a given pattern.

Given a pattern p at time i, we look for p in the layer sw_1, i.e. the smallest layer. If p is in S_{sw_1}, then all the occurrences of p in $T[i - sw_1 : i]$ belong to the class of the rightmost occurrence of p over T. If p is not in S_{sw_1}, then we look for any occurrence of p is in S_{sw_2}, the next layer in increasing order. If p is in S_{sw_2}, since it is not in S_{sw_1}, for the Prop. 3.1, any occurrence of p in this layer belong to the rightmost occurrence of p over T_i. Continuing in this way,

as soon as we found an occurrence of p in a layer, this occurrence correctly answer to REP(p). By using this trivial approach we can find REP(p) in time proportional to the length of the pattern times the number of layers.

Since many of the classic variable-length codes for integers, like the Elias's γ-codes, produce codewords of length proportional to the logarithm of the represented value, we can assume that the cardinality of SW is $O(\log |T|)$. Since that $|T_i| = i$, in the *online* fashion, we have that the time for REP(p) is $O(|p| \log i)$.

A similar result can be obtained by using a variant of the Amir et al. algorithm presented [1], but it does not support REP(LPF) operations in constant time and it does not support sliding window over the text.

Since Prop. 2.2 holds for the layers of our data structure, we can use the binary search to find the smallest layer containing a given pattern. Since $|SW| = O(\log i)$, for any classic variable-length code, the number of layers in our structure is $O(\log i)$ and the proof of following proposition comes straightforward.

Proposition 3.2. *Using any classic variable-length code, at any time i the multilayer suffix tree is able to answer to REP(pattern) for a given pattern in $O(|\text{pattern}| \log \log i)$ time.*

4 Solving REP(LPF) and REP(SPF)

As pointed out in the Senft's paper [18], the online suffix tree construction algorithms in [6, 11, 21]) keep track of the longest repeated suffix of the text. We use those information, one from any layer of the multilayer suffix tree, to answer in constant time to the REP(LPF) problem and the REP(SPF), where LPF is the longest previous factor and SPF is the set of prefixes of LPF.

Recall that, using Ukkonen's nomenclature, an Implicit Suffix Tree I_k, $1 < k < n$, for the text $T = a_1 \dots a_n$ is a Suffix Tree for the text $T[1..k] = T_k$. It is built starting from I_{k-1} by implicitly extending all the suffixes of T_{k-1} that are unique, i.e. those that are leaves, and explicitly adding the suffixes of T_k in decreasing order down to the longest suffix of T_k that is already in I_{k-1}, i.e. the longest suffix of T_k that appears at least twice in T_k. In order to efficiently perform the explicit extensions, the construction algorithm maintains at each step a reference to the longest repeated suffix of

the text. Let us say that at the end of the step k the longest repeated factor is $T[i \dots k]$. Eventually in some successive steps the longest repeated factor became $T[i+1 \dots k']$.

In order to solve the REP(LPF) problem in constant time, we let independently to grow each layer of the multilayer suffix tree to $I_{(k'-1)}$, where, for any layer, $k'-1$ is the last step of the construction algorithm which longest repeated factor starts in position i. This does not change the time complexity of the algorithm, since the overall steps for a text T are the same. In the mean time, in order to correctly represent the factors of offset b_α in each layer α, the sliding widow size are dynamically set to $SW_\alpha + \mathrm{LPF}_\alpha(i)$.

Now that any layer point out to the longest repeated suffix starting in position i accordingly to the layer size, we just take in constant time, along the building process, from any layer one occurrence of its longest repeated suffix. Then, we keep the value coming from the smallest layer with a pointed factor of length equal to the length of the biggest layer, i.e. $\mathrm{LPF}(i)$.

Proposition 4.1. *Given a text T of length n in the dynamic setting, at time i, the multilayer suffix tree of the text $T[1..i+LPF[i]]$ is able to solve REP(LPF) in constant time.*

Since $\mathrm{LPF}(i)$ is equal to the longest repeated suffix of $T[1..i+\mathrm{LPF}[i]]$ by the very definition of LPF, the proof of above proposition come straightforward from Proposition 2.2.

Concerning the occurrences of the set of prefixes of LPF (SPF for short), we are interested to those having a different bit length of their rightmost position, as the missing ones are just prefix of an occurrence of a longer one. With arguments similar to the ones used for the LPF case, we maintain inside the multilayer suffix tree a list SPF of (length, offset) pairs in the following way. At any time i, let occ be the position of $\mathrm{LPF}(i)$ found in the largest layer. The pair $(\mathrm{LPF}[i], occ)$ is added to the empty list SPF. Let LPF_α and occ_α be the length and the offset of the longest repeated suffix pointed out by the layer α. For any α in decreasing order if the last pair in the list SPF has length value equal to LPF_α, then this element is updated with the offset value occ_α. Otherwise the pair (LPF_α, occ_α) is appended to the list. Notice that the size of the list is smaller that or equal to the number of layers $O(\log i)$ as already assumed. In order to solve the REP(SPF) problem it suffice to retrieve a pointer to the internal SPF list of the multilayer suffix tree.

The proof of the following proposition is straightforward.

Proposition 4.2. *Given a text T of length n in the dynamic setting, at time i, the multilayer suffix tree of the text $T[1..i+LPF[i]]$ is able to solve REP(SPF) in constant time.*

At any time i, since any REP(SPF) value has *equal-cost* w.r.t. the rightmost occurrence of the corresponding LPF prefix, REP(SPF) can be used as dictionary pointers for the dictionary phrases matching the factors of the text starting at position i in any LZ77 based compression algorithm. Due to space constrain we omit the details of the next proposition.

Proposition 4.3. *Given a text T, using at any time i the REP(SPF) values provided by the multilayer suffix tree, one can correctly build the parsing graph $G'_{A,T}$ of the* Dictionary-Symbolwise Flexible Parsing *algorithm.*

We present some experimental results in order to evaluate the practical requirements in terms of space and time. We used as cost function just the length of the binary representation of numbers and then sliding windows of size equal to the increasing powers of 2 up to a fixed length *MaxDictionarySize* equal to 2^{24}. The space grows up to 35 Bytes times *MaxDictionarySize* for regular text file like the bible and the enwik8 ones (see Table 2 caption for more details), as our implementation uses 17 Byte in average per character for any layer. We compared the multilayer suffix tree (MLST) running

Input	Length	MLST	RMST	Δ
bible	4 MB	8.52 μs	1.77 μs	4.81
enwik8	100 MB	16.15 μs	3.00 μs	5.38
dna	104 MB	8.51 μs	2.19 μs	3.88

Table 2: Execution time table for different input. The bible file belongs to the Large Canterbury Corpus, the enwk8 file contains the first 100 MB of the English Wikipedia database and the dna file belongs to the Repetitive Pizza&Chili Corpus. The MLST column report the total time per character of the construction of the Multilayer Suffix Tree with max window size of 4 MB. The RMST column report the building time per character of one single suffix tree with sliding window of 4 MB. The Δ column contains the ratio between MLST and RMST times.

time with the running time of a suffix tree with sliding window of size *MaxDictionarySize* maintaining the rightmost occurrence of any internal node (RMST). At any time i, the position value of internal nodes in the path from the root to the point of insertion of the new leaf are updated. The average running time of the MLST is about 5 times greater than the RMST time, as shown in Table 2. Furthermore, since all the layers in the multilayer suffix tree are independent each other, it is easy to speed up the overall time by using a parallel handling of the layers. Moreover, since maintaining the rightmost position in RMST runs in $O(n \log n)$ average case and $O(n^2)$ in the worst case while the MLST run time is $O(n \log n)$ in the worst case, the use of the MLST is encouraged when the dictionary size become large and/or the text contains highly repetitive factors, e.g. using molecular biology data, as the Δ value for the dna file suggests.

References

[1] A. Amir, G. M. Landau, and E. Ukkonen. Online timestamped text indexing. *Information Processing Letters*, 82(5):253 – 259, 2002.

[2] M. Cohn and R. Khazan. Parsing with prefix and suffix dictionaries. In J. A. Storer and M. Cohn, editors, *Data Compression Conference*, pages 180 – 189. IEEE Computer Society, 1996.

[3] M. Crochemore, L. Giambruno, A. Langiu, F. Mignosi, and A. Restivo. Dictionary-symbolwise flexible parsing. In *IWOCA'2010*, volume 6460 of *Lecture Notes in Computer Science*, pages 390–403, 2011.

[4] M. Crochemore, L. Giambruno, A. Langiu, F. Mignosi, and A. Restivo. Dictionary-symbolwise flexible parsing. *Journal of Discrete Algorithms - IWOCA'10 Special Issue*, 2011.

[5] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. In *SODA '09*, pages 768–777. Society for Industrial and Applied Mathematics, 2009.

[6] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32:490–505, April 1989.

[7] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, Aug. 2005.

[8] R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *Data Compression Conference*, pages 302–311, 1995.

[9] S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. Compact directed acyclic word graphs for a sliding window. In *SPIRE*, pages 310–324, 2002.

[10] T. Y. Kim and T. Kim. On-line optimal parsing in dictionary-based coding adaptive. *Electronic Letters*, 34(11):1071–1072, 1998.

[11] N. J. Larsson. Extended application of suffix trees to data compression. In *Data Compression Conference*, pages 190–199, 1996.

[12] A. Lempel, S. Even, and M. Cohn. An algorithm for optimal prefix parsing of a noiseless and memoryless channel. *IEEE Trans. Inf. Theor.*, 19(2):208–214, Sept. 2006.

[13] Y. Matias and S. C. Sahinalp. On the optimality of parsing in dynamic dictionary based data compression. In *SODA*, pages 943–944, 1999.

[14] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304:87–101, July 2003.

[15] K. Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, Dec. 2007.

[16] D. Salomon. *Data compression - The Complete Reference, 4th Edition.* Springer, 2007.

[17] M. Senft. Suffix tree for a sliding window: An overview. In *WDS'05*, pages 41–46, 2005.

[18] M. Senft. Compressed by the suffix tree. *DCC'06, IEEE Computer Society*, 0:183–192, 2006.

[19] M. Senft and T. Dvořák. Sliding cdawg perfection. In *SPIRE '08, Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, 2009.

[20] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[21] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[22] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.